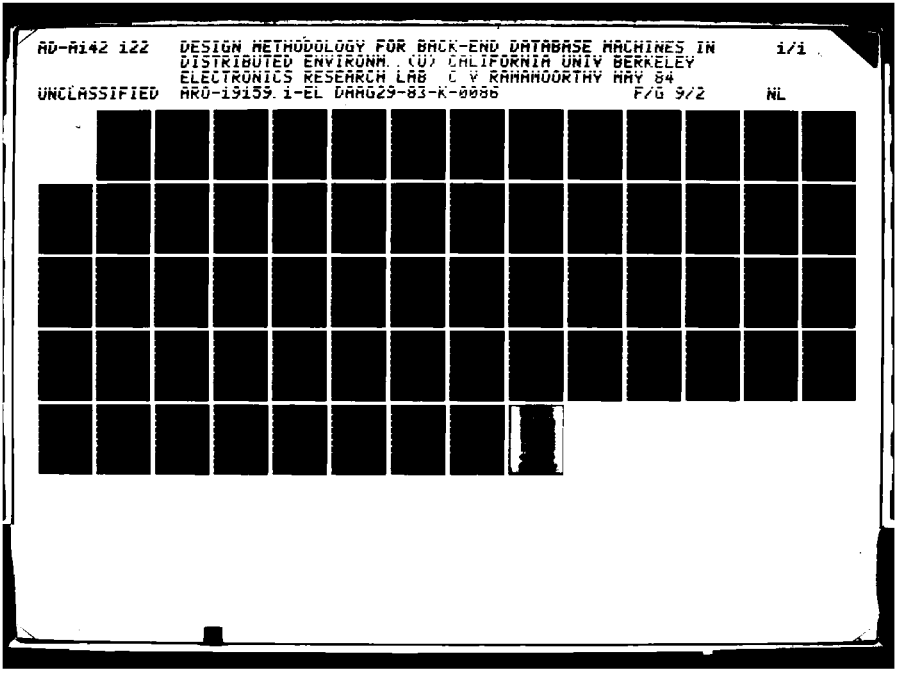
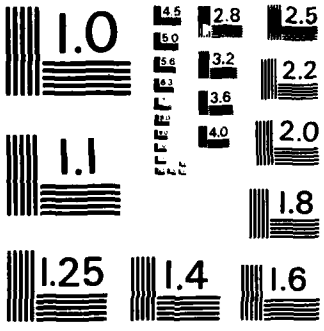

AD-A142 122 DESIGN METHODOLOGY FOR BACK-END DATABASE MACHINES IN DISTRIBUTED ENVIRONM.. (U) CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB C V RAMAMOORTHY MAY 84
UNCLASSIFIED ARO-19159.1-EL DAAG29-83-K-0086 F/G 9/2 NL



DESIGN METHODOLOGY FOR BACK-END DATABASE MACHINES IN DISTRIBUTED ENVIRONMENTS

TECHNICAL REPORT

C. V. Ramamoorthy

May 1984

U. S. Army Research Office

Grant DAAG29-83-K-0086

Electronics Research Laboratory
University of California
Berkeley, California 94720

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Design Methodology for Back-End Database Machines in Distributed Environments	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): C. V. Ramamoorthy	8. CONTRACT OR GRANT NUMBER(s): DAAG29-83-K-0086
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Electronics Research Laboratory, University of California, Berkeley, CA 94720	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709	12. REPORT DATE: May 1984 / 13. NUMBER OF PAGES: 60
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

NA

18. SUPPLEMENTARY NOTES

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Database Machines, Database Management Systems, Distributed Database Management Systems, VLSI, Distributed Processing, Concurrent Processing, Design Methodology

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The design Methodologies for database machines are introduced, both at virtual architecture and physical architecture levels. Starting from requirement specification, the architectures are derived through step-wise refinement and validation. Special attention is also paid to the impacts of VLSI technology.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

A-1

Design Methodology for Back-End Database Machines in Distributed Environments

Table of Contents:

Chapter 1 : Introduction 2
Chapter 2 : Design methodology for
Large scale Computer Systems 3
Chapter 3 : Functions of Back-End DB
Machines in Distributed Environments.10
Chapter 4 : Back-End DBMS Construction
from requirement Specification.....22
Chapter 5 : Classification and Comparison
of Database Machines 28
Chapter 6 : Impacts of VLSI Technologies
on DB Machine Architecture 37
Chapter 7 : Design of VLSI Database Machine
Part I - Methodology and a
Proposed Architecture 42
Chapter 8 : Design of VLSI Database Machine
Part II- Chip Design of Query Processor
from Requirement Specification
.....48
Chapter 9 : Current Status and Planned Work 56
References 58

Chapter 1. Introduction

1.1. Motivation and Objectives

Our overall objective is to develop a design methodology and to establish a basis for the design theory of the development of distributed processing systems. In order to give a concrete basis to our research, we have chosen a specific problem to study- the use of distributed computer systems for providing data management facilities in a node of an unreliable, secure network.

Current approaches to the design of distributed processing systems are based primarily on intuition and experience. As the computing power of processors increases with the development of VLSI technology, the size and cost of software and hardware design increases by leaps and bounds. The complexity of today's computer systems provides serious problems of maintainability, understandability, expandability and adaptability, which are only exacerbated by the trend towards multiprocessing and distribution functions. We need, therefore, a systematic approach for the design and analysis of distributed computer systems. To anchor our research in reality, we have chosen to develop our research around a subsystem of considerable current interest, a database machine backend for a node in a computer network. Although research on database machines has been comparatively recent, a sizable body of knowledge has been acquired concerning different aspects of them which we would like to systematize in a top-down approach. Special attention is also paid on the impacts of VLSI technology.

1.2. Overview of Contents

In chapter 2, we discuss the importance of design methodology and the particular one we deem appropriate for our research. Chapter 3 and 4 discuss the design of the virtual database machine architecture. Chapter 5,6,7,8 are devoted to the design of physical database machines. Both design considerations and design methodologies are given in these two closely related subjects. Chapter 9 gives the conclusion, current status and our planned work.

Chapter 2. Design Methodolgy for Large Scale Computer Systems

The complexity of today's computer systems provides serious problems of maintainability, understandability, expandability, and adaptability, which are only exacerbated by the trend towards multiprocessing and distribution functions. We need therefore, a systematic approach for the design and analysis of distributed computer systems.

Current approaches to the design of distributed processing systems (DPS) are based primarily on intuition and experience. These approaches cause expensive penalty : lengthy development time, unreliability, inability to tackle changing environment, etc. The methodology we will follow uses the concepts of abstraction, stepwise refinement, and modularity. To be more specific, system development is partitioned into stages and phases. The stages constitute a natural structuring based on major differences in applied technology. The phases, which make up a stage, impose an ordered, layered approach to design, reducing the risk of error and producing systems that are easier to understand and maintain.

2.1 Stages and Phases

2.1.1 Problem Definition Stage

During this stage the functional and nonfunctional requirements of the computer system are determined. We believe that successful system design proceeds from a clear understanding of the problem being addressed and, therefore, consider this stage to be of prime importance. Two phases of development occur during this stage to ensure the accurate definition of the problem: an identification phase and a conceptualization phase. The identification phase is informal and exploratory in nature. During this phase an identification report is produced that contains all available information on system responsibilities, system interfaces, and design constraints. The system requirements generated during the conceptualization phase contain (1) a conceptual model that formalizes the system's role from a user perspective and (2) the design constraints imposed by the application. The conceptual model is the standard against which system

functionality is measured throughout the design process.

2.1.2 System Design Stage

During this stage the hardware and software requirements are established for each component of the system. Requirements for the functional and nonfunctional capabilities of the components are specified, including the interfaces between these components (such as communication protocol, programming language support, and operating system primitives). These requirements are closely followed (1) during the procurement and design of the individual system components (such as computers, operating systems, and peripherals.) and (2) during the subsequent integration of these components into the final system.

The system design stage includes two phases : system architecture design and system binding. The main concern of the developer during the system architecture design phase is to investigate system design alternatives and their potential impact on the various system configurations being considered. In the case of a distributed database system the developer may use this phase to identify data and processing distribution and the number of nodes present in the network. Particular software or hardware components are evaluated to assure that a set of reasonable binding options exists.

During the system binding phase , the actual mix of hardware and software is selected . The hardware and software requirements generated during this phase may combine off-the-shelf and custom-built hardware and software components. How these components are selected is determined by the design constraints to be met and the available technology. Binding options are identified in the system architecture design phase, but the selection of specific components is done in the binding phase.

2.1.3 Software Design Stage

There are three phases involved in this stage. During the first phase, software configuration design, off-the-shelf software is procured and the overall high-level software system design for each programmable system component is established . This involves

(1) the structuring of the software into such division as subsystems, virtual layers , and tasks, (2) the definition of interfaces between components, and (3) the generation of requirements for each component. The program design phase takes these requirements and produces the program design (data and processing structures) , which together with all pertinent assumptions and constraints , makes up the implementation requirements. These are used by the coding phase to build the actual programs.

2.14 Machine Design Stage

This stage is similar to the first two phases of the software design stage. During this first phase of the machine design, the hardware configuration design phase, off-the-shelf machines are procured and high level architecture of custom hardware is designed. Component requirements are developed for all entities that are part of the custom hardware and passed on to the component design phase. A register-transfer level machine description which is used to determine the circuit design requirements and the firmware requirements, is generated during this stage.

2.1.5 Circuit Design Stage

Three phases of system development occur during this stage: switching circuit design , electrical circuit design , and solid state design. During each phase, design requirements are generated for the phase immediately following.

2.1.6 Firmware Design Stage

This stage consists of three phases that are an analog to program design, coding, and compilation. These phases are microcode design , micro-programming, and micro-code generation.

2.2 Steps

The ten steps listed below represent the design activities involved in each phase, regardless of the nature of that phase.

(1) formalism selection,

(2) formalism validation,

(3) exploration,

(4) elaboration,

(5) consistency checking,

(6) verification,

(7) evaluation,

(8) inference,

(9) invocation, and

(10) integration.

2.2.1 Formalism Selection

This step encompasses the activities involved in selecting a formalism for a particular problem domain. Each phase may involve one or more different formalisms : programming languages for coding ; pseudocode for program design; logic diagrams for switching circuit design; and stimulus-response graphs and logic models for conceptualization. Candidate formalisms are chosen on the basis of their expressive power in that domain and their ease of use, lack of ambiguity, ease of analysis , and potential for automation.

2.2.2 Formalism Validation

In this step, we determine whether a formalism has the expressive power needed for a particular task. We also evaluate how easy formalism are to use. These tasks may involve both theoretical and experimental evaluations. The validation step also includes evaluations of the formalism's potential for design automation and its ability to support

hierarchical specifications.

2.2.3 Exploration

This step encompasses the mental activities involved in synthesizing a design. These activities are creative in nature and depend on experience and natural talent. They cannot be formalized or automated unless the problem domain is significantly restricted.

2.2.4 Elaboration

In this step, ideas produced in the exploration step are given form through the use of various formalisms. Coding , specification writing, and circuit layout drawings are typical activities associated with this step , but it's scope extends to the building of a concrete object such as a piece of hard ware . The effectiveness of this step may be greatly increased through the use of a variety of design and manufacturing aids.

2.2.5 Consistency Checking

This step encompasses activities such as checking for incorrect use of formalisms; for contradictions, conflicts, and incompleteness in specifications; and for semantic errors. Checking includes verifying consistency between different levels of abstraction in a hierarchical specification and reconciling multiple view points.

2.2.6 Verification

In this step , we demonstrate that a design has the functional properties called for in its requirements specification. Since each phase has a requirement specification and produces a design , this step is equally important for all phases. A common example of this type of activity is verifying program correctness.

2.2.7 Evaluation

In this step, we determine if a design meets a given set of constraints . Constraints include both those that are part of the requirements specification for the phase and those that result from design decisions. The nature of evaluation activities depend on the type of constraints being analyzed. They include classical system performance evaluation of response time and workload by means of analytical or simulation methods; deductive reasoning for investigating certain qualitative aspects like fault tolerance or survivability.

2.2.8 Inference

In this step, the potential impact of design decisions is assessed. Questions addressed are (1) How will the system impact the application environments ? Can we afford the implementation? Is personnel retraining too expensive ? ; (2) Can subsequent phases accommodate the decisions made in this phase ? Is the bandwidth choice reasonable ? ; (3) How does the design affect our ability to maintain and upgrade the system ? Will parts be available five years from now ? ; and (4) How does the design affect implementation options ? Is there a good reason for ruling out mainframes ? These issues must be considered in every phase, but they are particularly critical in stages that define architectures.

2.2.9 Invocation

This step encompasses the activities associated with releasing the results of the phase. It includes quality control activities involving tangible products and review activities that lead to the formal release of output specifications. The release of output gives the step its name, since this release in effect invokes subsequent phases.

2.2.10 Integration

In this step, the portion of the total system designed in the phase is configured and tested. Traditionally integration is considered a design area , and would therefore qualify as a stage in the framework. However, we have chosen to distribute integration

activities among the phase because (1) the expertise needed to test a portion of the system is similar to the expertise needed to create its requirements, (2) the assumptions made in a phase about the nature of the products that could be delivered by subsequent phases must be checked once the subsequent phases complete their tasks, (3) all models used to make these assumptions must be validated, and (4) errors found during integration must be resolved in the phase that created the requirements.

2.3. Goals:Using above mentioned hierarchical approach, we believe we can develop a design methodology that will be able to :

(1) provide an evolving system with controlled expansion.

(2) represent effectively and efficiently the decision making constraints by a specification language.

(3) provide a means for incorporating design alternatives and trade-offs at various design steps and design levels.

(4) provide design attributes and documentation for evolution (growth and modification) so that changes can be made without reconsidering the whole design process.

Chapter 3. Functions of Back-End DB Machines in Distributed Environments

Database sharing is one of the main advantages a network environment provides. However, several problems which we do not encounter in a monolithic system arise:

- How should data be distributed?
- Will the whole system still operate if one node fails?
- How to authenticate users from remote sites?
- How to reduce the communication overhead?
- How to coordinate tasks between several sites?
- How to recover a failed node?

In this chapter, we look into these problems, give a brief survey of existing solutions, and add our comments.

3.1 Environments

A general distributed environment may consist of several networks, each connected through gateways. Each network may have different characteristics from others; e.g., topology, communication medium, and the physical distance between two nodes, etc. Eavesdropping may occur on the network and malicious users may try to break into the system. Communication links may be broken at any time and any node can fail. Users at different sites run programs independently, and they may want to access the data base at the same time - delete, update, read, append, etc. All these factors contribute to the complexity of a Distributed Data Base Management system (DDBMS).

3.2 Functional Requirements of a DDBMS

In this section, we examine functional requirements of a DDBMS. We focus especially on those functions that are brought up by the distributed nature of our target environments.

3.2.1 Data Distribution & Replication

Required at each autonomous site are two kinds of data: frequently accessed and less frequently accessed. Frequently accessed data should be stored locally. The object of data distribution is to satisfy each site's needs in an efficient way. If two or more sites frequently access the same data, then the data should probably be replicated, under some tradeoff considerations. Several advantages of replication can be identified:

(1) Higher availability.

(2) Better response time.

(3) Reduces communication traffic.

(4) Load balancing.

However, this is true only when most of the accesses to the replicated database are read requests. For update requests, all the advantages go away and several problems arise. A list of update strategies that tend to solve these problems may be found in [Li79]. We discuss here only two common strategies:

(1) *The Unanimous Agreement update Strategy:* In this scheme, unanimous acceptance of the proposed update by all sites having replicas is necessary in order to make a modification, and all of those sites must be available for this to happen. In this design, the availability of a replicated file for update requests is (1-p)**N, if there are N copies; p is the probability that a node fails.

(2) *Single Primary Update Strategy:* Update requests are issued to the primary replica, which serves to serialize updates and thereby preserve data consistency. Under this scheme, the secondaries diverge temporarily from the primary. After having performed the update, the primary will broadcast it to all the secondaries some later time. Availability of this scheme is (1-p); again, p is the probability that a node fails.

3.2.2 Authorization

A multi-user data base system, whether distributed or not, must permit users to share data in a controlled and secure manner. Problems encountered in centralized databases, which have to be shared, inc .de authorization validation, creation and destruction of tables in a dynamic manner, etc. When the system becomes distributed, new issues crop up. One of the main issues is that of security of data while it is on the communication medium. This is a problem which is purely an outcome of the distributed nature of the environment. For some applications the confidentiality of data is critical, and we ought to have mechanisms to prevent *data theft* by techniques such as wiretapping. In recent years this issue has aroused much interest in researchers leading to substantial development in the field of cryptography. Use of cryptographic techniques for the safety of transmitted data entails the following:

The sender *encrypts* the data to be transmitted using a key, yielding *cipher text*.

The cipher text is transmitted over the insecure channel. This is safe because even if someone were able to record this data, its meaning would be unintelligible to him.

The receiver *decrypts* the cipher text to get back the *clear text*, i.e. the original data.

Currently DES is a very popular encryption mechanism which is based on what in literature is reffered to as the *conventional key* encryption scheme. An inherent drawback of this scheme is its inability to provide the facility of *digital signature* in a simple way. Recent researches have brought to light an alternative encryption scheme known as the *public key* encryption system. This mechanism solves the problem of implementing digital sinatures in a simple and elegant manner. One unfortunate aspect of this scheme is the current availability of fast enough technology to implement this scheme in an eficient manner. However, this problem is not inherent in the scheme, and we hope technology will develop fast enough to overcome it.

Other authorization problems include those of password verificatio, access control, etc. These are problems which are not due to the distributed nature of the system. Good

and efficient solutions to these problems abound in literature.

3.2.3. Protocol Handler

To coordinate executions among remote sites, we must design a set of protocols for DDBMS; e.g., locking protocol, recovery protocol, etc. The details of these protocols will be discussed later. In this section, we discuss how to design new protocols in an automated way to guarantee their correctness.

Protocol synthesis is a process of designing new communications protocols. The objective of developing automatic protocol synthesizers is to provide a systematic way of designing protocols such that their correctness can be ensured. Although protocol analysis methods are useful to various extents in validating existing protocols, they do not provide enough guidelines for designing new ones. What protocol designers need is some set of design rules or necessary and sufficient conditions, so that their designs are guaranteed to be correct. The newly designed protocols need not go through the analysis stage to be checked for their correctness.

We developed a protocol synthesis procedure which constructs the peer entity from the given local entity which is modeled by a Petri net.[Do83] If the given entity model satisfies certain specified constraints, the protocol generated will possess the general logical properties which a protocol synthesizer is looking for. The synthesis procedure is very general. It is applicable to every layer of the protocol structure.

To construct the desired peer entity model, there are three tasks which should be conducted in sequence :

(1) Check local properties of the given local entity model to make sure that it is well-behaved. This can be done by generating and examining the structure of its state transition graph.

(2) Construct the peer state transition graph from the above generated state transition graph according to some well designed transformation rules.

(3) Construct the peer entity model in Petri nets from the peer state transition graph.

3.2.4 Transaction Management

The transaction management system is responsible for scheduling system activity, managing physical resources, and managing system shutdown and restart[Gr78]. Transaction is a unit of consistency and recovery. We could divide a transaction into three phases:

(1) Read Phase: In this phase, access to data objects must be authorized.

(2) Execution Phase

(3) Write Phase: In this phase, transaction may be aborted or committed.

Concurrency control mechanisms may be used to solve problems in Read phase and Recovery management may be employed to solve problems in Write phase. The Execution phase will be discussed in a later section.

3.2.4.1 Concurrency Control

Concurrency is usually introduced to improve system response time. However, if several transactions run in parallel, the system may be left in an inconsistent state unless accesses to shared resources are regulated. There are three forms of inconsistency:

(1) Lost Updates: Write -> Write dependency.

(2) Dirty Read: Write -> Read dependency.

(3) Un-repeatable Read: Read -> Write dependency.

Note that reads commute, so we don't have Read -> Read dependency.

There are basically two ways for solving concurrency control problems. One is by locking mechanism, and the other uses timestamp-based protocols.

3.2.4.1.1 Lock Management

We could define consistency in terms of lock protocols. We say that a transaction T observes the consistency protocol if:

(a) T sets an exclusive lock on any data it dirties.

(b) T sets a share lock on any data it reads.

(c) T holds all locks to EOT.

An important issue is the choice of lockable units. It presents a tradeoff between concurrency and overhead, which is related to the granularity of the units themselves. For fine lockable units, concurrency is increased, but it has the disadvantage of many invocations of the lock manager, and the storage overhead of representing many locks. A coarse lockable unit has the converse cases. It would be desirable to have lockable units of different granularities coexisting in the same system.

Another important issue the lock manager must deal with is deadlock. There are several ways to handle this problem:

(1) Timeout: causes waits to be denied after some specified interval. It is acceptable only for a lightly loaded system.

(2) Deadlock Prevention: by requesting all locks at once, or requesting locks in a specified order, etc. One generally does not know what locks are needed in advance, and consequently, tendency is to lock too much in advance.

(3) Deadlock Detection and Resolution: Deadlock detection problem may be solved by detecting cycles in wait-for graphs. Backup process is handled by Recovery manager.

3.2.4.2 Recovery Management

The job of recovery manager is to deal with storage and transmission errors. There are three possible outcomes of each data unit transfer:

(1) Success (target gets new value)

(2) Partial failure (target is a mess)

(3) Total failure (target is unchanged)

The recovery manager must be able to back up to a consistent state no matter what failures occur in order to keep the data integrity. Since recovery management is a critical part of reliable DDBMS, we will discuss it in length in this section. Following is a brief survey of existing mechanisms to support fault tolerance for a transaction processing system.

3.2.4.2.1 Transaction Commit

If several copies of a data are distributed around the network, then they must all be kept up to date to avoid inconsistency. The following technique called *Transaction Commit* can be applied: We select a primary, or originator site. It will serve as the coordinator. It first sends update requests to other sites, then waits for their answers. If everyone agrees to participate in the updating, the coordinator sends the commit request and everybody does the actual updates at that time. Note that after it agrees to participate, no site can change its mind any more, and during the updating, the data must be locked, i.e., no other user can access the data. To guarantee that the mechanism will work under any single failure, we still need two supplementary mechanisms, *shadow pages* and *audit trail,* which are discussed in the next two subsections.

3.2.4.2.1.1 Shadow Pages

The idea of shadow page is that before the transaction is committed, all updates should actually go to a shadow copy of the original data, so that when crash occurs, we can still recover the data to the original consistent copy. This is different from the "multiple copies" in that one of the two copies here is kept only temporarily, and after the transaction commits, the original copy is deleted. The system must keep track of where these shadow pages are, and must be able to remove all of them when the system recover from crashes.

3.2.4.2.1.2 Audit Trail

If some failures occur during the transaction processing, either after or before the transaction commits, the recovered site must be able to identify which state it is in. If the transaction already commits, it must replace the original copy by the shadow copy. If the failure occurs before it receives the commit request but after it agrees to commit, then it must check other sites to see whether the commit action is already taken by other sites; if it is, it performs the commit operations, otherwise, removes the shadow copy. If it has not agreed yet, then apparently the transaction must already aborted, so it can remove the shadow copy. To keep its state, each site must records the sequences of actions on its data. However, the audit trail itself may be damaged. To keep the integrity of the audit trail, another form of multiplication called *stable storage* [St80] may be used.

3.2.4.2.1.3 Stable Storage

The basic idea of stable storage is "write twice". We always keep two copies of the data, and always update them in the fixed order; first primary, then secondary. If while writing the primary, the system crashes, we copy the secondary to the primary. On the other hand, if when writing the secondary, the system crashes, then we can copy primary to secondary. Now, a natural problem arises, how do you know which state you are in? We can not rely on another audit trail, because the problem would become circular. An easy solution is to use checksum to check the integrity of the data. If one is bad, the good can be copied to it. If both are good, the crash must occur just after we successfully write the primary copy; in this case, the primary should be copied to the secondary.

3.2.4.2.1.4 Software faults

One thing that is usually ignored intentionally in designing reliable systems is software faults. Most systems assume that there is no bug in system programs. However, the catastrophe caused by software faults happens everyday in the world. Since

they are easily ignored, they also go undetected during execution, thus making recovery very difficult, if not impossible. Recently a lot of concern has been shown about this problem. Good references can be found in [Ki84]. Here we mention only the idea of *recovery blocks:*

For each block of code, we introduce alternate blocks which perform the same function but with different algorithms and different degrees of precision or complexity in a hope to make things work despite failure of one method. To detect a software fault, an *acceptance test* is performed, which checks the validity of the results generated by the code block. The acceptance test keeps the integrity of the data. If the results fail to pass the test, then recovery mechanism is initiated. The process state must be reinitialized before entering the code block, after which an alternate block is selected and the execution starts over again. If all blocks fail the test, then error is reported.

A potential problem of this mechanism is that the coding effort is substantially larger than that without recovery blocks. However, for critical applications, if the software is rather complex, sustantial savings in terms of debugging efforts could be achieved. However, this technique can only deal with software faults, and we did not mention here how the state is saved and recovered.

3.2.4.2.1.6 What Else?

In the above discussion, we enumerates many fault tolerant techniques that are related to transaction processing. Although they are not the whole story, they identify most of the important mechanisms that we feel should be included in a distributed, transaction processing system. However, there is one thing we haven't discussed yet, i.e., how are cooperating processes recovered from crash occurring in one of them? *Domino Effect* may occur when we try to back up these processes to a consistent state. We devote the next section to investigating this problem.

3.2.4.2.2 Achieving Fault Tolerance Using Message Passing

Coupled with the development of distributed computation, message-passing has become the primary candidate for an operating system kernel structure. One of the important functions that can be achieved via message passing is system reliability. Although research is still under way, it is generally believed that, at the cost of redundancy, message-based systems are able to yield fault tolerance.

There are many software-controlled schemes for reliability. Among others, *checkpointing* and *transaction*, the two we discussed above, are most fundamental. Incorporating these schemes, more specific techniques have been designed and applied to real-world environments [Ba81], which features the concept of *process pairs*, have proved to be of practical value. The idea of *publishing* [Po83], as has been simulated in Demos/MP, an experimental distributed operating system currently under development at Berkeley [Po84], is a simple and powerful tool for tolerating faults on an Ethernet. The Auros§, a Unix-like operating system being implemented on the M68000-based multiprocessor Auragen 4000, introduces the novel notion of multi-way message backups and periodic synchronization, which looks very promising.

Fault-tolerant operating systems always need the support of multiple processors, either in a distributed or a tightly-coupled fashion. Traditionally, the cost-effectiveness was not attractive except for some specific and defense-oriented applications. With the advent of VLSI, the situation has reversed almost over night. Highly-reliable systems have finally reached such application domains as airline reservation, banking, etc. with reduced expense. This section focuses on some of the important issues considered by Non-Stop, Publishing and Auros.

---

† both trademarks of Tandem Computers Inc.
§ a trademark of Auragen Systems Corporation.

3.2.4.2.2.1 Types of Faults Tolerated

Assumptions about the environments differ from system to system. With regard to faults, most message-based systems commonly assume the following.

1. A message-based fault-tolerant system is able to tolerate single hardware faults. Software failures are not handled.
2. Failures must be *detectable* and *non-deterministic*. In other words, failures must be *recoverable*.

3.2.4.2.2.2 Duplicated Resources

The major concern here is the manner in which duplicated resources are used to provide fault tolerance.

In Non-Stop, the idea of process pairs is implemented as follows. The requester and the server both keep a process backup respectively. The checkpointing is performed at a very fine grain. Whenever a primary process receives a message, it checkpoints its backup. If the primary crashes, the backup takes over and when the old primary recovers, it becomes a backup. During its recovery period the new primary is not checkpointed. Each message is identified by a unique sequence number. Redundant operations are avoided by comparing message sequence number and an internal log kept by each process.

Duplication in Demos/MP is restricted to a centralized recorder that records every message flow over the Ethernet. This recording activity is called publishing. The recorded information is categorized according to process-id. To cut down the work during the recotlyery stage, occasional checkpoints are performed. Processor state since last checkpoint is also kept in the recorder. If the recorder crashes, a second one will be elected. The recovery procedure is a standard roll-forward discipline: Firstly restore state, then replay interactions since checkpoint and lastly discard outputs since failure time.

The Auros system extends Non-Stop's process-pair idea one step further to yield a scheme known as multi-way message transmission. In Auros, every message sent by the

sender to the requester goes to three places: (1) primary destination, (2) backup destination, and (3) sender's own backup (increment a counter, actually). (1) and (2) are the analogy of a process pair whereas (3) serves primarily for the purpose of preventing redundant messages be from being resent. Every process interrupt is checkpointed. But the interrupts by kernel in backup checkpoints are not checkpointed.

Whenever the primary has read a system predefined number of messages, the primary and its backup are synchronized. Again, like every checkpointing mechanism, this is for performance considerations instead of reliability. Without checkpointing, the reliability can still be achieved, but the efficiency of recovery will be degraded.

3.2.4.2.2.3 Crash Detection

To detect crashes in Non-Stop, the following steps are taken:

N1. Every second, each processor sends an unsequenced acknowledgement packet over each bus to every processor.

N2. Every two seconds, every processor checks whether it has received an unsequenced packet from each other processor.

As far as crash detection in Demos/MP is concerned, a recovery manager is implemented. Two types of crashes are handled by the manager.

D1. A process crash causes a trap to kernel, which stops the process and sends a message to the recovery manager containing the error type and process id of the crashed process.

D2 To detect processor crashes, the recovery manager spawns a watchdog process in the recording node. If no messages have been seen in a while, the processor is considered to have crashed and is restarted. To avoid the watchdog's misjudgement, each processor is required to send out null messages from time to time even if it has nothing to say.

Since Auragen is still under development, it is not clear at this moment the specific mechanisms used for crash detection. Since the Auragen 4000 is an architecture of several clusters of multiprocessors, it can be predicted that failures local to a cluster is

detected locally and periodic polling by a global server on each cluster is necessary.

3.2.4.2.2.4 Transparency

The Non-Stop process pairs is a mechanism that is not transparent to the user. An application program desiring fault tolerance must explicitly manage the process pairs. This burdens the application programmer very badly. A side effect of this situation is the large amount of application-specific code, which makes transporting packages difficult. Also, the upgrade of system may cause some software compatibility hardships. It has been reported that the situation is being improved.

Both Demos/MP and Auros feature transparency of high reliability. Users of these systems have no idea about the realization of fault tolerance. To us, this is essential. Furthermore, since publishing is a centralized mechanism as opposed to the other two decentralized ones, it causes less perturbation during recovery.

Chapter 4. Back-End DBMS Construction from Requirement Specification

Our ultimate goal is to provide a computer aided interactive design environment for designing distributed systems. DDBMS is our first design instance. In our view, the scenario of future design process should look like following:

> *A user wants to design a new system. Although he knows what are required in his system, he may not be able to solve the conflicts between requirements; he knows some design issues, but he is unable to manage the complexity of all possible design tradeoffs. Therefore, he asks for help from the computer. He works interactively with the computer to get a consistent requirement specifications. There may be several alternatives given by the computer for each design problem, the user can ask the computer for explanation of certain choices; he may randomly select one, or he may specify further requirements to narrow down the scope of possible alternatives, e.g. lowest cost, highest reliability,..etc. Finally, if everything is agreed between the user and the computer, the latter will start working on a full report in details of what are required for lower level components of the system. The whole design process may be iterated several times for different levels of the target system architecture.*

Now we will see how this strategy could be applied to our DDBMS design. Before getting into the details, we shall point out that, not every DDBMS is built from scratch; there may already be some physical constraints ; e.g., network topology may be fixed, supporting operating system may not be changed, etc. These can be integrated into the requirement specifications. Also, the design may even stop at some earlier phase; there is a trend to design DDBMS based on existing database systems with fixed physical designs[Ro84]. In this case, the design must stop at logical access path optimization.

We see the whole design of a back-end DBMS as a 3-phase process.

Phase A: User requirements -> Network design + Node requirements

Phase B: Node requirements -> Subnetwork design + processor requirements

Phase C: Processor requirements -> (HW + SW) design + bus structure design

Our primary interest lies at the level of Phase B. However, we shall start with phase A because many design decisions must be done at this level. We do not plan to get

into phase C at this stage, but the output of phase B should be feasible in terms of processor constraints or realizability. This chapter will be devoted to the discussion of our methodology and the design of phase A, later chapters will discuss phase B in details.

4.1 Consultation System

All design phases start with requirements from upper levels and produce requirements for lower levels. There are two kinds of requirements: functional and non-functional. For phase A, functional requirements are usually the same in all DDBMS's, because basically they all have the same problems to solve; the difference only lies in the degree of functionality. However, non-functional requirements may be drastically different among different enterprises. Some important concerns are: system cost, throughput, response time, expandability, adaptability, user friendliness, and reliability. To meet both functional and non-functional requirements in designing a DDBMS is certainly a very difficult job. Design methodology must be employed to save design effort.

To help users and designers specify their requirements, we are currently building a consultation system. The idea is to store current expertise of DDBMS into a database using Ingres[St76] facility, and to employ an interface program to retrieve the information so that users may reference this expertise and remove inconsistency or infeasibility of their requirements. Our first version is very primitive because the validation of requirements can not be done automatically yet, the system only provides some hints to help users make the right specifications.

We are investigating the knowledge organization of the consultation database. It should consist of the following categories of information to answer questions of users or designers in the future:

(1) What are the requirements that need to be specified for a DDBMS?

(2) What are the design issues that need to be considered?

(3) How many alternatives do we have for solving a particular problem? What are the advantages and disadvantages of each?

(4) Descriptions of the current state of existing DDBMS's. How do they solve different problems? Any measurement data or evaluation reports available?

(5) Is there any preference relations for design options? How to check the conflicts between two requirements?

We need some tools to help us build up the consultation database and other elements of future design environment. These will be described in the following sections.

4.2 Requirements Language

We must provide a requirements specification language so that users can state their requirements naturally yet rigorously enough to avoid ambiguity. We are currently extending RSL[Al77] notions for our purpose. Basically, there are five types of objects:

(1)*Elements*: Analogous to "nouns" in English. Elements are objects and ideas which the requirements analyst uses as building blocks for his/her description of the system.

Examples:

- ALPHA: the class of functional processing steps
- DATA: the class of conceptual pieces of data necessary in the system
- R-NET: the class of processing flow specifications

(2)*Attributes*: Analogous to "adjectives" in English. It formalizes important properties of the elements. Each attribute has associated with it a set of values, e.g., mnemonic names, numbers, text strings.

Examples:

- INITIAL-VALUE for DATA
- INPUT for ALPHA

(3)*Relationships*: Analogous to "verbs" in English. It corresponds to the mathematical definition of binary relation, a statement of an association of some type between elements. Relationship is non-commutative, i.e., a

subject element and an object element are distinct.

Examples:

- ALPHA INPUTS DATA

- or DATA is INPUT to an ALPHA, where INPUT here indicates the relationship

(4)*Structures*: There are two structures:

- R-NET (SUBNET) structure: identifies the flow through the functional processing steps (ALPHAs) and is thus used to specify the system response to various stimuli.

- VALIDATION-PATH: is used to specify performance of the system.

(5)*Segment*: It consists of groups of element types, relationships, attributes, and structures which arise from some underlying issues of requirements definition. There are five segments:

- DATA segment: the logical relationship among pieces of information and interaction of the information with the rest of the system.

- ALPHA segment: the basic processing steps in the description of a set of functional requirements.

- R-NET segment: the specification of the flow of processing steps

- VALIDATION segment: the definition of the performance requirements to be met by the system.

- MANAGEMENT segment: information necessary to support the disciplined management of a requirements engineering project.

4.3 Requirement Analysis

After users have specified their requirements, some requirement analysis must be made to remove inconsistencies among requirements. In this section, we discuss two tools that may help the analysis.

4.3.1 Payoff Measures and Payoff Trees

Payoff measures are measures of "goodness" and are thus associated with the non-functional attributes(reliability, performance, etc.) of the system. The development of the payoff measures was guided by the need to make them compatible with the partitioning approaches. There are several ways to partition a system:

(1) partitioning by hierarchical component level(DDBMS, nodes/ internodal network, computers/intranodal network, hardware/software).

(2) partitioning by level of abstraction(functional, virtual, physical), which is superimposed on each of the component levels of the first type of partitioning.

(3) partitioning by data processing software component(application system, operating system).

(4) partitioning by subsystem affinity.

The payoff measures were developed along the lines of the first type of partitioning. Furthermore, they are applied essentially at the virtual(to guide the allocation process) and physical (constraining physical design) levels of abstraction of the second type of partitioning.

In order to use the payoff measures for guiding the design path at each stage, each payoff measure must be decomposed into the lower-level, design-controllable factors which determine the value of the payoff. This decomposition, called the *payoff* tree is structured hierarchically along hardware and software lines. To obtain a high-level payoff value, the contributions to the payoff from its subtree are aggregated based on analytical models or simulation.

We have selected a set of payoff measures and developed payoff trees appropriate to the design of DDBMS's. The development is made so as to facilitate the evaluation of the payoff.

4.3.2 Preference Graphs

The Function Option Library(FOL) in our consultation database is organized in the form of preference graphs. In different decision domains, the designer must choose among the options available as the process of function refinement or elaboration is carried out. These domains are represented in the FOL. As the design process proceeds through the successive phases, the design path through the preference graphs will be traced out to greater and greater depths.

Chapter 5. Classification and Comparison of Data Base Machines

To develop an effective methodology for designing distributed backend database machines requires in-depth knowledge about the target itself. Chapter 3 has covered issues of distributed systems in general. The next chapter will focus particularly on the impact of VLSI. This chapter discusses the current status of backend database machines. The rationale behind database machines is first examined. Being unable to support efficient database operations, the deficiencies of conventional computer systems are highlighted. As remedies to these deficiencies, there have been a number of database machines proposed, a representative subset of them is classified according to a simple taxonomy. What are the guidelines in designing a database machine? Some suggestions are given with a real-world example. Finally, problems faced by these machines are also investigated.

5.1. Background

People's desire increases proportionally with the power they acquire. The introduction of general-purpose databases has stimulated a great demand for a higher performance data management capability. A direct consequence of this demand is that many installations have reached the point of resource saturation. The explosive increment of data is certainly responsible for this crisis. But if we take a closer look, the most essential point is not that we are *unable* to handle a large amount of data but that the performance of handling this data is severely *degraded* due to its "large" quantity. The implication, therefore, is that system structure must be the primary source of causing this performance degradation. On the one hand, the operating system may be inadequate to support efficient data retrievals and updates. On the other hand, it may be the case that the underlying system architecture itself is deficient in supplying fast operations needed by very large databases.

As pointed out by [St81], the problems of operating system support for DBMS are many folds. For example, operating system buffering is sometimes redundant because a DBMS has to buffer anyhow, so why bother to double the costly operation? In terms of

disk prefetch and replacement, the DBMS usually has a more accurate guess than the operating system because the former knows better which block will be used next. Furthermore, crash recovery is a central issue in database systems, especially for those in distributed environments. But since an operating system does not guarantee the committed block be written back to disk immediately (due to its buffering), crash recovery in such a system becomes very hard.

There are yet many other concerns in [St81] such as data segment sharing, context switch overhead, convoy effect, etc. What they ended up doing in Ingres was to modify the Unix kernel by adding those features which they considered essential and deleted those they thought redundant. The purpose: to achieve a satisfactory database performance. The major advantage of this approach is its relatively low cost. However, if the quest for an improved performance is higher, one cannot escape from facing embedded bottlenecks in the system.

An alternative to an upgrade is the offloading of database management functions from an existing computer to a backend machine which handles nothing but database operations. This approach, as surveyed in [Ma80], is the software realization of DBMS on dedicated conventional computers. It is clear what it buys is host's load relief at the cost of some extra hardware. Since the backend acts as a database "machine", the frontend will be able to run more jobs yielding a better global throughput.

The disadvantages of this approach come from the loose coupling of host and backend. Since there is no shared memory, additional overhead may be introduced due to the inevitable copy operations as part of the now necessary communication between the two parties. More importantly, the inherent deficiencies of von Neumann machines are generally ignored by these systems. The limitations of conventional von Neumann architectures in terms of DBMS support are the following:

(1) The familiar von Neumann bottleneck: large quantities of data need to pass through the processor-memory channel of a limited bandwidth.

(2) The sequential nature of address decoding in traditional memory technology.

(3) The uniprocessor architecture.

(4) The lack of intelligence in secondary storage.

In fact, these issues are not unique to database systems only. Researchers in the computer architecture community have long been trying to do something about them. As far as DBMS is concerned, this often means a specially devised architecture is used as the backend rather than what is implemented in systems belonging to the previous category. A taxonomy of database machines based on their different departures from von Neumann model is given next.

5.2. Taxonomy

A database machine can be categorized according to a combined consideration of (1) the approaches taken to eliminate or reduce the above limitations or bottlenecks, and (2) the main contributions it makes [Su84]. A more detailed guideline follows:

I. *Intelligent secondary storage devices.* Techniques that fall into this domain try to process and manipulate data where they are stored. Typical methods include context to addressing mapping, content to content, hardware garbage collection, etc.

II. *Filters and intelligent controllers.* Main memory to secondary storage bandwidth utilization can be increased by either reducing the amount data that passes through or by using multiple channels. The so-called data filters and intelligent disk controllers are designed for this purpose.

III. *Associative processors.* The goal of introducing associative memory is to eliminate the limitations of traditional memory's sequential decoding behavior. However, associative memory is still a very expensive device under current technology.

IV. *Multiprocessor database computers.* Three more subclasses can be distinguished:

(i) *Loosely-coupled distributed systems.* Interconnection in these systems are usually bus- or tree-oriented. Data is processed in a distributed fashion (not geographically distributed, however) to achieve some performance upgrade.

(ii) *Loosely-coupled backend machines.* The host down-loads its database functional specialization to a backend machine. This machine can be a conventional computer which inherits software capability from its host only. This is essentially the approach mentioned earlier. The other possibility is to build a novel piece of hardware designed solely for database applications.

(iii) *Dynamic MIMD processing.* This is by far the most innovative database machine technology. Techniques like multiprocessor cache, dynamic processor allocation, data flow, switchable memory modules are potential candidates.

(iv) *Special purpose processors.* These processors make use of VLSI technology and offer more promise than others.

Notice that under this taxonomy, such approaches as the one taken by Ingres are not considered as database machines. In other words, if the system hardware configuration is not altered, even though virtually equivalent functionalities are achievable, a conventional system is not classified as a database machine. As an exercise of the taxonomy, a number of real-world database machine are summarized in table 5-1.

5.3. Design Considerations

Design considerations are closely tied with the development of methodologies. To implement a database machine, the first criterion is to identify user requirements. The attributes listed below, as devised by [Ep80], provide a coarse discipline:

I. *Transaction rate.* The designer must understand what is the maximum transaction rate expected by the user. The speed of memory devices chosen must be based upon the the required rate. It is trivial to see that the faster the memory the higher it costs.

II. *Storage requirements.* What is the largest possible amount of data the user will be storing? A very large amount of data exceeding a certain threshold value implies that single level storage no longer suffices; instead multiple levels of storage must be implemented.

General guideline	*Finer grain technique*	*Technology/System*
Intelligent secondary storage devices	—	bubble memory, CASSM, RAP, RARES
Filters and intelligent controllers	Filtering	CAFS, VERSO, SURE
	Mass memory controller	DBC
	Data modules	DIALOG
Associative processors	—	STARAN, NON-VON, ARM
Multiprocessor database computers	Loosely-coupled distributed systems	Bus: MICRONET, MDBS Tree: HYPERTREE, REPT
	Loosely-coupled backend systems	IDM, iBDP
	Dynamic MIMD processing	DIRECT, SM3
Special purpose processors	—	systolic arrays (CMU) join processor(Maryland)

Table 5-1: *A taxonomy of a number of database machines*

III. *Access patterns.* If the commonly used access manners are predictable, then the database machine can be tuned toward its optimal state with regard to these patterns. This is reasonable because certain applications reference data by certain

fixed key values.

IV. *Costs*. The costs a user can afford certainly precludes everythingelse.

The cost/performance ratio can be computed using these attributes. This ratio serves as a market probe which helps in defining an initial target. For example, IDM's decision was to focus on mid-scale users. The so-called "mid-scale" users are those that require a transaction rate of less than 1K but faster than 0.1K per minute. Also, it is assumed that certain access patterns can be observed. Specifically, IDM is designed to achieve transaction rates as high as 2K/min or as low as 100/min. It can store up to 32 Gigabytes of data. A family of IDM's, spanning a wide range of performance from mid-scale towards low-ends, are implemented. Unlik the commercial market, a military application may have a narrower view in how this cost/performance ratio should be used in making design decisions. However, we see no particular situations where the principles reported here are not applicable.

There are other fundamental tradeoffs that need be considered. We consider the following:

Access methods

The choices are *complete scan* or *hashing/indexing*. The former is a linear algorithm and is in most cases impractical because of its terrible time complexity. However, if multiple moving disk heads are available, for some small databases this approach may be a win. The latter scheme is typified by B-trees, ISAM, and hashing, etc. They are algorithms with a complexity that is proportional to the *log* of the number of cells in searching; rather than the number itself. Tremendous amount of time can be saved by these methods without the help of multiple search elements. If access patterns can be predicted, the second scheme can outperforms the first at a much lower cost.

Storage Medium

It must be determined whether fixed head disks are sufficient. If not, a complete line of moving head disks, each offering a different cost/performance ratio, are available. Depending upon the user's specification, the designer can decide which to choose, at an

optimally low cost/performance ratio. A noticeable fact is that the price per unit storage drops as the size of the system expands.

Processors

Another decision the designer has to make concerns the processing elements. There are quite a few off-the-shelf microprocessors readily available. He/she must decide if they are functionally sufficient, given that the only task this processor has to perform is database operations. One possible conclusion he may draw from a careful study is that most OEM processors are computation bound, which do not always match database requirements. As a result, the designer sometimes ends up building special-purpose processors uniquely for database machine support.

Cache

A disk cache can be implemented using random access memories. By the argument of locality of reference, a disk cache pays only if a block is referenced more than once in a short period of time. Again, a careful cost/performance measurement must be considered. The cache size and speed is highly dependent upon secondary storage size, secondary storage speed, and most importantly, the user requirement. If desired, a relatively high cost implementation of cache can yield an effective disk access at close to main memory speed.

Control flow

A very important issue is that of the flow of control in transactions. The choices are (1) single thread: only a single transaction is allowed to execute at any moment, or (2) multiple threads: multiple transactions are permitted, the system provides a scheduling mechanism similar to that in a multiprogramming environment. This issue is tightly coupled with the choice of disk controllers. If the function of single thread or multithread control is realized at the controller level, the tradeoffs are minimal. However, if the function remains in the central processor, then the operating system has to take care

of the non-trivial scheduling among transactions provided a multi-thread approach is taken.

Functionality

The functionality of a backend database machine can be abstracted as the following seven levels. (1) cache controller, (2) search unstructured files, (3) record management system, (4) basic relational data management system, (5) backup and recovery facilities, (6) protection and data definition facilities, and (7) full user support. The higher the level of abstraction the more work is offloaded into the backend machine.

5.4. Summary

Suppose queries types can be partitioned into *overhead-intensive* and *data-intensive*, under the condition that queries are overhead-intensive, the database will not be cost-effective [Ha79]. It is further shown that data-intensive queries can be performed efficiently on database machines if the function performed on the data is a function the database machine provides [Ha82].

From these performance analyses, it is clear that putting in the right functions into a database is the premise of the success of a database machine. Users may have a variety of requirements and, thus decisions must be made based upon the specification and the available technology. The purpose of this chapter is to provide a first order survey of the current status of database machines. These ideas will serve as a bottom line in later phases of the development of distributed database machine design methodologies.

Chapter 6. Impacts of VLSI Technologies on DB Machine Architecture

VLSI technology means having upward of many hundreds of thousand switching devices on a single silicon chip with feature size approaching one micron. In the last two decades, IC technology has advanced from a few to tens of thousands of transistors on a single silicon chip. For the first 15 years, since the inception of ICs, the progress in making ICs for every complex structure has moved in an exponential fashion, at the rate of doubling the number of transistors that could be placed on a single chip every year [Mo79]. Although the growth rate has slowed down to doubling every 18 to 24 months over the last few years, physical limits suggest an ultimate density improvement of 1000 times over that attained by today's IC technology. [Ri80].

With the large number of switching elements available in a single chip as promised by VLSI technology, the question that arises naturally is: What can we do with this technology and how can we best utilize it? In what follows we will discuss several architectural trends based on VLSI technology, particularly those related to the data base machines.

6.1. Concurrent Processing

Although concurrent processing has been touted as a major area of extensive research for a long time, its importance was not widely recognized until recently as more and more applications requiring high degree of concurrency have come forward.

The impact of VLSI technology towards this direction is two-fold:

a) Intra-chip Concurrency:

For the last few years, several self-contained, single-chip computers have been implemented. As technology is moving rapidly, we expect that it will soon be possible to have multiple processors, memory modules, other logic blocks. and communication paths on a single wafer. While pin limitation is generally accepted as one serious drawback of VLSI technology [Bl77], this evolution provides more flexible as well as more tightly coupled computers with low communication cost and high bandwidths. This intra-chip concurrency, together with the decreasing cost of hardware, invokes the following trends:

1) Migration of Functions - VLSI Implementation of Algorithms:

Hardware implementation of an algorithm is always faster than a software implementation. of hardware. Further, by using more hardware components, additional speed-up can be obtained. For example, using a linearly connected network of size O(n), both the convolution of two n-vectors and the n-point discrete Fourier Transform can be computed in O(n) units of time, rather than O(n logn) as required by the sequential FFT algorithm. The design cost of VLSI chips to implement algorithm is designed carefully in the first place [F080]. The complexity of designing special-purpose chips is about the same as designing a high-level algorithm for the same problem if the underlying algorithm is "good" and a proper methodology which transforms the good algorithms into a final layout in a more or less mechanical way is used. As there are sizable problems in data base area can be solved by good algorithms, the design of special-purpose VLSI chips to implement them becomes feasible and cost effective. [Ku80]

2) Concurrent Processing of Tightly-Coupled, Asynchronous Software Modules:

For computer systems in which parallel-processing is achieved through separate IC chips, the communication remains a major and difficult problem. The advantage of parallel processing may sometimes be overshadowed by the excessive communication required among the modules, particularly for those which are tightly coupled and highly interactive. The difficulties are due to the following two factors: First, because of different loading capacities, the delay-power product of a connection residing within a single IC chip is much smaller than those interconnecting separate chips. At present, the ratio is more than two orders of magnitude and will become larger as proper scaling of MOS circuits leads to faster and smaller circuits which operate at lower power levels. Hence, bringing a signal from one chip to another results in a significant performance penality, either as increased power assumption or as exorbitant delay. The second factor is due to the pin limitation, for which only a few interconnection schemes may be applied among chips and most communications can only be achieved on the time-sharing basis.

All above difficulties disappear, however, if intra-concurrent VLSI chips become feasible. As multiple building blocks and communication paths can be built in one chip, more freedom is obtained in arranging them and most importantly, at a much lower communication cost. As for DBMS applications,. an immediate example is that the four-process structure in INGRES[St76] can be implemented as four asynchronous hardware modules and operates in the pipelining manner, more users(queries) can thus be active at the same time.

b) Inter-Chip Concurrency

By exploiting the advantages of VLSI technology where a complex CPU, a single-chip microcomputer, or even an intra-concurrent single chip microcomputer might cost only a few dollars, it becomes economical to design a highly concurrent system using a multiplicity of microcomputers providing more processing power than would be possible or practical using a single microcomputer.

In summary, these two levels of concurrency provided by VLSI technology suggest a general computer architecture of the form depicted in figure 6.1.

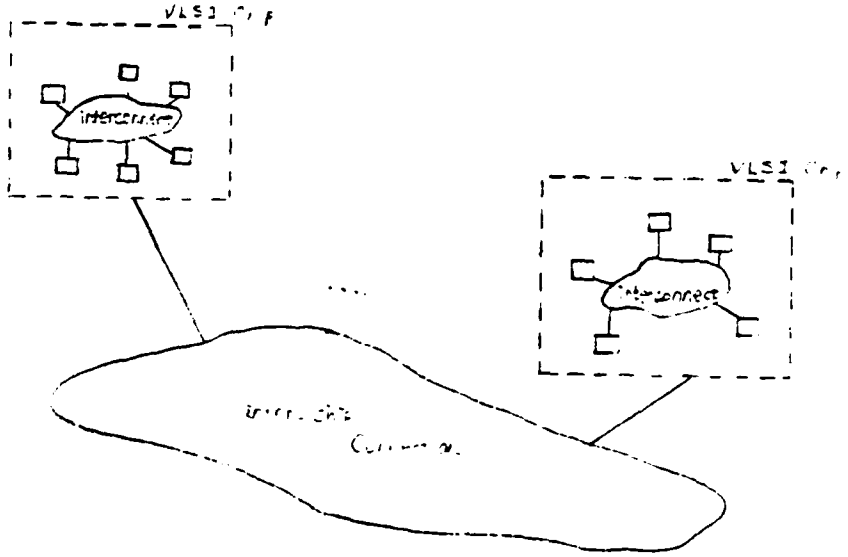


Figure 6.1

6.2. Customisation

Custom logic, predominately in the form of gate array today, offers the system designer important advantages over standard products. It allows the designer to implement his/her own architecture exactly. It is due to the Mead-Conway design method [Me80] that computer scientists can experiment with his/her own custom circuits. Since then a series of new generation of VLSI circuit design aids emerge. A partial list include the Caesar [Ou81] Circuit Editor, MacPitts datapath generator [Su82], Esim and NL/RNL [Ba80] design-rule checkers and switch-level simulators, as well as TV[Jo83] and Crystal [Ou83] timing analyzers.

One major advantage of custom logics, in addition to implement those good algorithms, is to tailor the architecture towards certain operating environments. Indeed, although most DBMSs are designed for general purpose, the average user demands (requirements) vary from system to system. It has been proven [De81] that no existing database machine architecture is optimal for all the demand patterns. For those systems in which average use demands are known, customized architecture is certainly required to optimize the performance (where, of course, general algorithms are still provided). As for database applications, the average user demand patterns can be short queries, aggregate queries, or multiple-relation queries. On the other hand, characteristics of relations (e.g. average length of the relation, average hit ratios, etc) may as well serve as design factors for customized logic.

6.3. Distributed Intelligence in System Components

Because of the large number of switching elements in the VLSI chip, it is possible to introduce more intelligence into different components of a computer system, such as memory and I/O processor, so that the processing load on the system can be distributed more evenly among the different modules of the system, and at the same time reducing the amount of communication among the different components. One example in this area is the design of a fast cellular associative memory which expands the functions of conventional associative memory [Le79]. Another example is the enhanced-logic memory

which is also an extension to associative memory aimed at the VLSI technology [De79]. The design of a more powerful I/O processor, such as the Intel 8089, has been a step in this direction.

Chapter 7. Design of VLSI Data-Base machine

Part I - Design Methodology and A Proposed Architecture

7.1. Introduction

After the virtual architecture has been instantiated, it needs to be implemented. While the derivation of virtual architecture is based mainly on functional requirements, as we discussed in Chapter 4, the implementation phase considers mostly performance and cost-effectiveness requirements.

The design of architecture for a virtual system is greatly influenced by technology. Today, with the low cost of hardware and advances in communication media, the distributed computer system has become the dominating architecture. The merits of distributed system are that they provide high throughput, modularity, reliability, availability, and reconfigurability, with relatively low cost. Along with the advantages of VLSI technology, in this chapter we will also present the design methodology of constructing a distributed database machine which is composed of multiple, interconnected VLSI chips from user performance requirements. A particular architecture will also be introduced.

7.2. The Design Methodology

Successful computer architectures are usually the result of many months of careful planning and development. Such an intensive planning effort requires an integrated design methodology that cover the entire architecture development life cycle. Furthermore, this methodology must be specialized to the particular application, technology, and organization.

The architecture design methodology we propose includes, based on our discussion in chapter 6, two steps: Global inter-chip architecture design and Local chip architecture design. Each of these two steps, in turn, has two phases: architecture analysis and architecture binding. The main concern of the developer during the architecture analysis is to investigate various design alternatives satisfying the performance requirements and come

up with several candidate designs with certain preference index. During the architecture binding phase, the most preferred candidate is selected and technology constraints are checked. If, unfortunately, the proposed architecture is not feasible according to the current technology, the next promising candidate is selected and the architecture binding phase restarts again. In the case there are no more candidates available, the commitment made at previous stage will have to be invalidated and the process restarts from the architecture binding phase of the previous stage. In the worst case there is no more global, interchip architecture available to commit, the performance requirement is deemed to be unfeasible. The whole architecture design process is depicted in figure 7.1

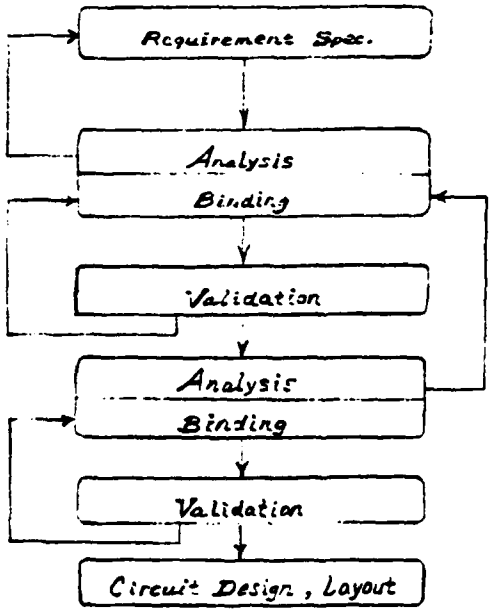


Figure 7.1

7.3. Global Interchip Architecture Design

The global inter-chip architecture design stage basically implements the process of partitioning, by which subsystems and subprocesses instantiated in the virtual architecture design can be grouped into different sets (hopefully, chips), and global interconnection. Due to the relatively large communication overhead among physical modules, the objective of partitioning is to group the subsystems into implementable modules with a

minimum amount of inter-module interactions, that is, the modules are loosely coupled. The basic design issues are:

1) How many partitions are appropriate?

2) Which virtual modules should go to which partition?

3) How are those partitions interconnected?

4) Which partitions should be implemented as custom modules?

5) How much intelligence should be distributed, as we discussed in section 6-3?

All above decisions should be made according to the following performance requirements:

1) In a multi-programming environment, the desired degree of multiprogramming.

2) The average workload of the system.

3) The desired average, best case, and worst case response time at average workload.

4) The cost constraint.

Both analytical and simulation studies will be conducted to resolve the above highly-interrelated design issues. Following guidelines are used during the study:

1) Heuristics are used in coming up with the candidate architectures and queuing analysis will be conducted to compare various alternatives. Based on [Ba75], requests to the system will be typed and functional modules will be classified. Execution Speed for the operating modules will be assumed, and thus serves as the requirements for the chip design stage, and are validated in later stages.

2) Although the problem of finding the optimal partitioning that minimizes the interaction is NP-complete and some heuristics, e.g. max-flow min-cut technique [Ra78], have been proposed, no consideration has been paid to the preservation of concurrency under partitioning. Good heuristic will be studied.

7.4. A Proposed Global, Inter-chip Architecture

Although our study towards formal derivation of global, inter-chip architecture has just been initiated, the study of chip design can be carried out as long as a clean interface exists between these two stages. In this section we will introduce a particular global, inter-chip data base machine architecture which is derived intuitively based on appropriate justifications. Indeed, this architecture may be deemed as the output derived from our incoming formal approach under certain particular workload and requirements.

7.4.1. Overview of the Proposed Architecture

When operational, the complete system will be composed of four main components: a host processor, a single-chip back-end database controller(DBC), a set of query processors (QP), a set of local disks together with their corresponding intelligent disk controllers (DC). The DBC is directly coupled with the host system and is connected to all the QPs through the local Bus (LB). As we assume that a back-end distributed database environment exists, the DBC is also connected to other DBCs of the distributed database.

The technology we assume is that VLSI can provide tens of processors, tens of memory modules, and several i/o ports, on a single chip such that certain amount of intra-chip concurrency can be explored. Through the interconnection network, the set of QPs are connected to the set of DCs and each DC may directly access some of the on-chip as well as off-chip memory modules. An overall picture is shown in Figure 7-2.

Software-wise, we assume that a suitable version of UNIX and INGRES[St76] exist such that they are suitable for this inter-/intra- chip concurrent environment. The host processor will handle all communications with the users and all queries are down loaded to the back-end DBC. The DBC, with the modified INGRES and some global information about the database(e.g. system catalog) residing, will parse the incoming query, modify it according to integrity control, decompose it into a sequence of one-variable operations, and finally form a query packet to be transmitted to one of the suitable QPs for execution through the Local Bus.

If the query is coming from some other site of the distributed data base, the DBC should handle all the communication function (e.g. protocol management, encryption/decryption) before the "true" query processing can start. The query processor will, after receiving the query execution packet, generate a strategy such that some optimal concurrency can be achieved, bring in the related pages of the relation(s), do the operation(s), and return the result packet back to the DBC. Concurrency Control is implemented at DCs and locking is used at page level. One file/relation is assumed and the file is not segmented across local disks.

7.4.2. Rationale of the Design

The following features best characterize the above architecture and differentiate it from other current, existing DBM architectures:

1) Concurrency: The inter-query concurrency is achieved through the intra-concurrent nature of the DBC and multiplicity of QPs . On the other hand, intra-query concurrency is achieved by the intra-concurrent QP on which the query execution packet is executing. While most existing DB machine architectures explore intra-query concurrency by cooperating different QPs (e.g. DIRECT[De79]), this architecture relieves the DBC from excessive control and produces more tightly coupled concurrency through distributed, local QPs.

2) Customization: As each QP may be customized, it can be designed tailored to some particular, average query pattern(s) without sacrificing its generality. As we will discuss in next chapter, this architectural optimization also relieves the DBC from excessive, software emulated planning and provides better performance.

3) Distributed Intelligence: Due to the discussion in section 6.3, the function of DCs are expanded. We think concurrency control should be implemented as close as possible T`o where data is stored, for the following reasons:

   a) Too much traffic will be generated between DBC and QP (or, among QPs) if page locking is done at DBC (or, at QP)

b) Higer the granuality of locking lowers the degree of data sharing.

c) Distribute the responsibility of concurrency control increases the degree of data sharing and reliability.

7.4.3. Remarks

It should be emphasized here that no commitment is made to the proposed architecture for all the applications. As we have mentioned in the beginning of this section, the architecture we just proposed may only be suitable for certain particular environments. The sole purpose here is to illustrate the impact of VLSI technology on DB machine design and serve as an example for our chip design discussed in the next chapter.

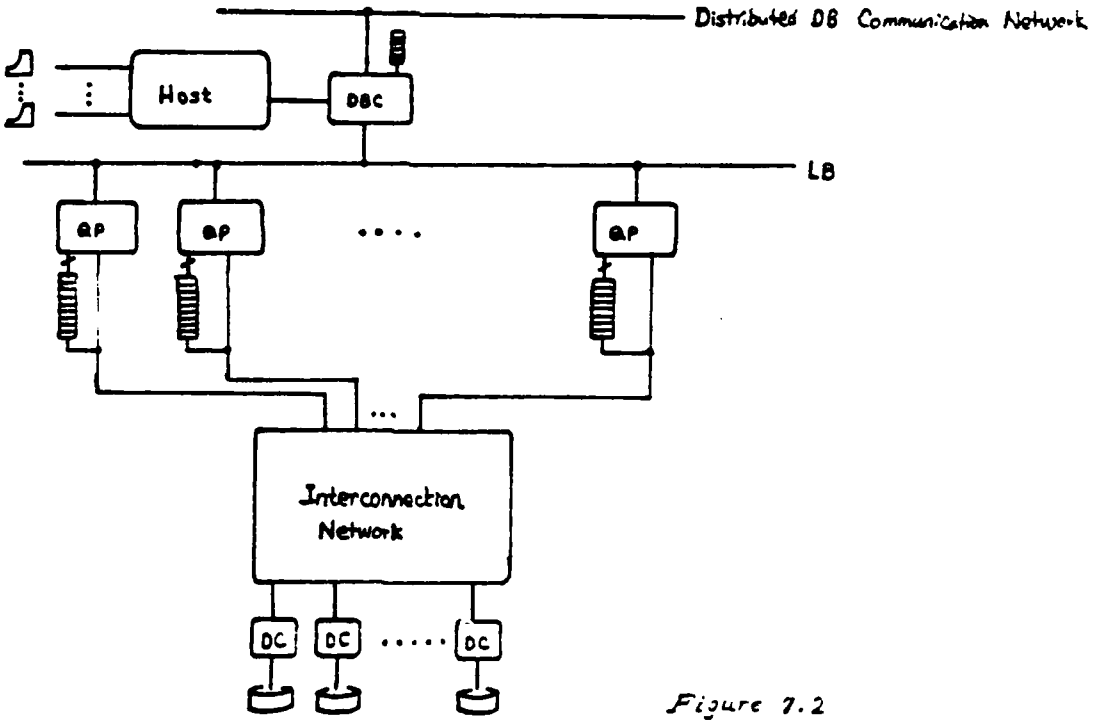


Figure 7.2

8. Design of VLSI Database Machine

Part II - Chip Design of Query Processor from Requirement Specification

8.1. General Philosophy

As we have discussed in Chapter 6, VLSI technology provides us with a chance of customizing our own logics under relatively low cost compared with conventional LSI technology. This is particularly useful for those systems whose typical operating environments can be estimated. Fortunately, most data base systems fall in this catagory.

The target we are interested in particularly is the query processor single-chip architecture, which we assume to be intra-concurrent in nature.Its functions were introduced in Chapter 7. While general query-handling algorithms are equipped with each QP, we decided to design the chip architecture based on certain user request characteristics such that the performance is optimized for those typical, average request patterns. To be more specific, our strategy here is to sacrifice some "boundary" operations such that we have better chance in achieving optimal performance. For systems in which there exist several competing patterns, compromise must be made for balance.

Being strongly favored by general VLSI design methodology [Me80], we believe that the design should be as modular as possible. In the architecture analysis phase, alternative architecture models should be selected while some of the design parameters, e.g. size, ratio,etc. may remain open. These parameters are instantiated at architecture binding phase where constraint/performance tradeoffs are resolved.

8.2. Classification of Queries

In [Ha79],[Ha82] three classes of relational queries are identified : overhead-intensive, data-intensive, and multi-relational queries. Here for discussion purpose we classify queries according to the number of relations involved and its aggregate/non-aggregate nature. To simplify the discussion, we assume that the typical, average queries in the target system are one or two relational, non-aggregate and are of the form:

operation(relation_attributes) where

qualification 1

.

.

qualification n .

and qualification i may be single-relational qualification or double-relational qualification

Let NSQ = set of {qualification1,.....,qualification n}

NSQL1 = set of single-relational qualifications on R1, the first relation involved.

NSQL2 = set of single-relational qualifications on R2, the second relation involved.

NDQL = set of double-relational qualifications on R1 and R2

thus |NSQ| = |NSQ1| + |NSQ2| + |NDQL|

Note that for one-relational queries, |NSQL2| = 0 and R2 does not exist.

The above query is then assumed to be transformed into the following sequence:

1) retrieve into temp1 where NSQL1

2) retrieve into temp2 where NSQL2 (if |NSQL2| != 0)

3) (if |NSQL2| != 0) join temp1 and temp2 into temp3 based on NDQL

4) operate on temp3 (if |NSQL2| = 0 then temp3 = temp1)

8.3. A Proposed Architecture for One-Relational Queries

Although several architectures are suitable for one-relational query processing, we propose here a particular one for analysis purpose and its architecture is depicted in Figure 8.1. This architecture is designed in favour of the execution of one relational queries with one or two qualifications. The sequence of operations activated for a one-relational,

two-qualification query is as follows:

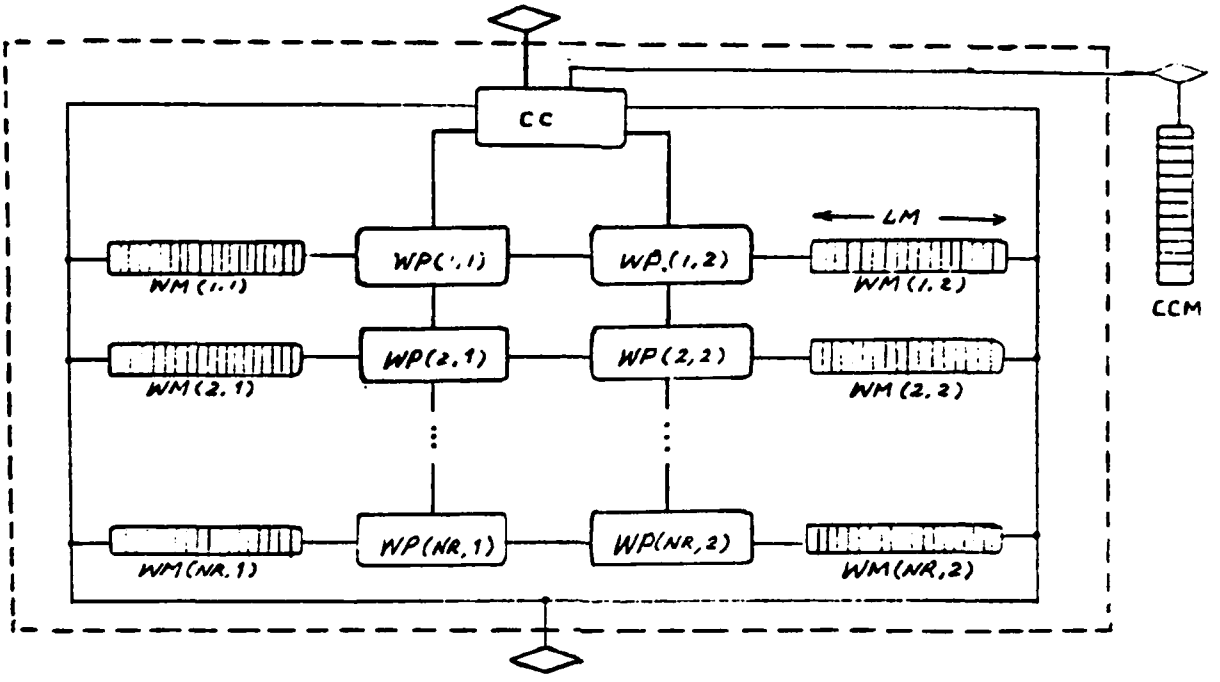


Figure 8.1

1) The Chip-Controller (CC) determines what functions should be done by the Working Processor(WP). It then broadcasts certain control messages to WP(i,j), 1<=j<=2, 1<=i<=NR, to activate the appropriate functions in WP(i,j) and instantiate certain parameters(e.g. number of cycles of comparisons). At the same time, the CC issues a request to the appropriate Disk Controller(DC) , according to the information in the query execution packet, through the Chip- Global-Bus(CGB).

2) The appropriate Dc, after allocates the relation file, loads WM(i,1), 1<=i<=NR, NLT(=LM/LT, see (4)) tuples(records) sequentially.

3) The CC then initiates a global operation cycle. During this cycle WP(i,1) fetches the next tuple in WM(i,1), compares it with argument 1 and passes the tuple, together with the result of comparison, to WP(i,2). WP(i,2) does the second comparison and if both the results passed from WP(i,1) and WP(i,2) are true then the

tuple is written to WM(i,2). After each of WP(i,j) has done NLT comparisons, the global comparison cycle terminates, and the results collected in WM(i,2), $1 <= i <= NR$, are collected by the CC (and are stored in the off-chip CCM).

4) Let

NT = Number of tuples in the relation

LT = Length of tuple in bytes

LM = Length of the array memory module in 1K bytes

if $(NT*LT)/(NR*LM) = NL > 1$, then the sequence (2)~(3) will be repeated NL times. Each cycle is preceded by a DC request from CC.

The design tradeoff in this architecture is that increased NR increases the amount of parallelism but also increases the total circuitry required by the WPs. This will greatly reduce LM and more disk accesses are required. To determine the optimal NR and LM, the following analysis should be carried out:

Let

STRATE = data rate from DC to CCM (s/byte)

DAVAC = average disk access time (s)

DRATE = data rate to QP (s/Kbyte)

WPCOMP = average time of a WP to do a comparison

a = ratio of circuitry required by a WP to 1K bytes of on-chip memory

BCOM = average time required to communicate with DBC

1KCOMP = circuitry complexity required by 1K bytes of memory

r = total circuitry constraint

b = ratio of circuitry required by the CC to that of 1K bytes of memory

Then the required response time for the given average single-relational, two qualification queries is :

response(NR,LM) = BCOM + NL*(DAVAC + NR*LM*DRATE + (!+(LM/LT))*WPCOMP + a*NT*LT*STRATE

and the total circuitry required is estimated to be :

complexity(NR,LM) = 1KCOMP*(b+ 2*NR*LM + 2*a*NR)

Our objective then is to minimize response(NR,LM) under the constraint that complexity(NR,LM) <= r. The minimal response is then compared with the required response time to determine if the above plan is feasible. Techniques like Lagrangian Multipliers can be used to solve the above algebric equation easily.

8.4 A Proposed Architecture for Two-Relational Queries

According to the transformation process suggested in Section 8.2, concurrency can be obtained if we could process on separate relations concurrently before join is required. The architecture of query processor we propose for this class of queries is an extension of what we proposed in Section 8.3 and is shown in Figure 8.2.

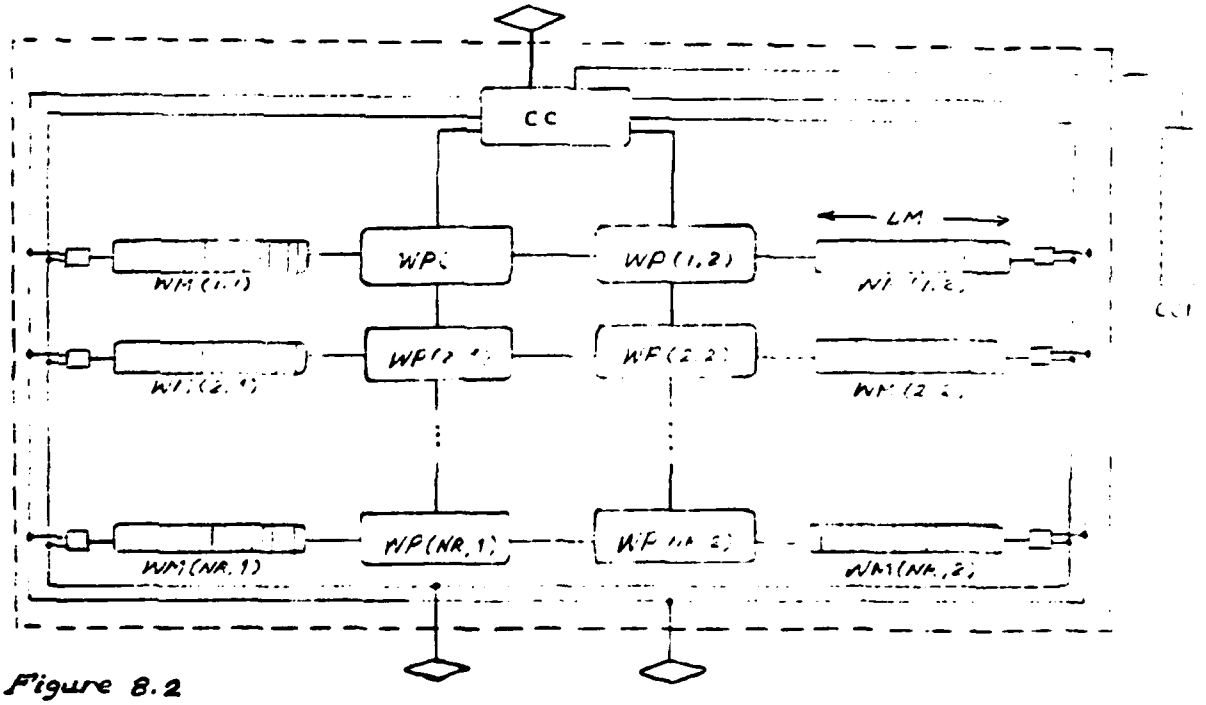


Figure 8.2

The difference of Figure 8.□ from Figure 8. is that instead of having only one CGB and one I/O port to DC, two of each are provided. This modification ensures two parallel paths for relation loading from the DCs. The working environment we assume is that the typical, average queries are two-relational with two single-relational qualifications for each relation and one double- relational qualification.

The proposed system, when receiving an incoming typical, average pattern query, works as follows:

1) The CC determines the sequence of operations should be executed.

2) The CC, once identifies the DCs associated with the relations, issues separate control messages to respective DCs for tuple loading. The amount of data and the destinations are also included in the control message.

3) The DCs, after allocating the associated relations, start loading the first chunk of data to the working memories. If these two relations reside on different disks, these tasks can be done in parallel. Otherwise, the loading is sequential.

4) The WP-WM pairs are partitioned into two parts. The first part works on the first relation tuple selection and the second part works on the second relation. Once the tuples are exhausted, a complete message is sent to the CC and the CC stores the results in the off-chip CCM.

5) When both parts finish execution, the CC could ask for another loading if there are still some unprocessed data remaining and the sequence (3)~(5) repeats.

6) After all the single-relational tuples are selected, the CC determines the inner and outer relations[De79] used for joins. It then distributes the qualified tuples, which are stored in CCM, evenly into the left half of the WMs(here we assume that one distribution is sufficient).

7) The WPs are then doing the join parallelly. The results of join are stored in the right half of WMs.

8) The tuples are then selected, based on the double-relational qualification, in the right to left manner.

The analysis for the target architecture is carried out as follows:

Let

WPJOIN = average time of a WP to do a two-tuple join

NTi = number of tuple in relation i, 1<=i<=2

LTi = length(in bytes) of tuple in relation i, 1<=i<=2

ai = hit ratio of single-relational selection of relation i, 1<=i<=2

b = fraction of NR rows assigned to relation i in the single-relational selection stage, 1<=i<=2 $\doteq$ (NT1*LT1)/(NT1*LT1 + NT2*LT2)

NR1 = NR*c

NR2 = NR*(1-c)

NLi = NTi*LTi/NRi*LM , 1<=i<=2

Then(assume relation 1 is the outer relation in join)

$$\text{response}(NR, LM) = BCOM + MAX\{(NL1 * (DAVAC + NR1 * LM * DRATE + [1 + \frac{LM}{LT1}] * WPCOMP + a1 * (NT1 * LT1/NL1) * STRATE), (NL2 * (DAVAC + NR2 * LM * DRATE + [1 + LM/LT1] * WPCOMP + a2 * (NT2 * LT2/NL2) * STRATE)\} + (a1 * NT1 * LT1 + NR * a2 * NT2 * LT2) * STRATE + [(a1 * NT1 * LT1)/NR] * (WPJOIN + WPCOMP)$$

$$\text{complexity}(NR,LM) = 1KCOMP * (b + 2*NR*LM + 2*a*NR)$$

8.5. Architecture Design for Multiple Typical, Average Query Patterns

The analysis in Section 8.3 and 8.4 are based on some particular query patterns. The style of analysis, indeed, is applicable for any single pattern. In case there are multiple, competing patterns, however, some compromise should be made for optimal performance.

Assume there are N competing patterns, each of which occurs with probability u(i), 1<=i<=N. The response time for each of them is response(i,NR,LM). Our objective is then to minimize

$$\sum_{i=1}^{N} u(i)*response(i,NR,LM)$$

such that complexity(NR,LM) is less than or equal to r, where

complexity(NR,LM) = 1KCOMP * (b + 2*NR*LM + 2*a*NR) in our particular example.

8.6. Concluding Remark

Although we have paid special attention to the next generation VLSI chip design, it should be noted that what we have developed in this chapter is general enough for conventional technologies also. As an example, the chip architecture can actually be implemented by LSI logics and the complexity constraint can be replaced by cost constraint, fan-out limitation, etc.

Chapter 9. Current Status and Planned Work

The work we have done in this project can be summarized as follows:

1) It can been seen from Chapter 3 and Chapter 5 that we have acquired sizable knowledge and have done a great deal of analysis on the subject of database machine architecture, both at virtual and physical levels. Some innovative contributions have also been made by ourselves to have the knowledge more complete.

2) It has been shown in chapters 4 and 7 that we have successively developed the meta-level methodologies for the virtual and physical database machine architecture design.

3) We have paid special attention to the impacts of VLSI technology to the physical database machine architecture design. As can be seen from chapters 7 and 8, the design methodology we developed is applied successfully to the global as well as local architecture design of database machines. Some appropriate architectures are also proposed.

4) We have investigated the approaches appropriate for the design automation of this particular subject. As we could see from chapter 4, a consultation system combining both software engineering and artificial intelligence techniques are currently being built.

What we are planning to do at this stage include:

1) Our research on distributed data base design will be continued. In the next stage we will also pay special attention on upgrading database systems to knowledge base systems. Combined impacts of VLSI technology and knowledge engineering will be addressed.

2) A complete knowledge base supporting our design methodology will be constructed. The combined effort of software engineering and general problem solving techniques will be elaborated.

3) Global Architecture analysis and derivation, as we discussed in chapter 7, will be further conducted based on formal performance evaluation. The strategies of reasoning on various architectures will be developed.

4) Requirement specification languages for various levels of architecture will be designed and tested.

References

[Al77] Alford, M., "A Requirements Engineering Methodology for Real-Time Processing of Requirements", *IEEE Trans. Software Eng.*, vol. SE-3, No. 1, Jan. 1977.

[Ba75] Baskett, K.M. et al." Open, Closed, and Mixed Networks of Queues,," pp248-260, April, JCACM, 1975

[Ba80] Baker C.M. and Terman C., " Tools for Verifying Integrated Circuit Designs," Lambda, Fourth Quarter, 1980, pp22-30

[Bl77] Bloch, E., and Galage, D. "Component Progress: Its Effect on High Speed Computer Architecture and Machine Organization," in High Speed Computer and Algorithm Organization by Kuck, S., Lawrie, D., and Saneh, A. (eds.) New York: Academic, 1977

[De79] Dewitt, D.J., " DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems", IEEE TC , June, 1979, pp395-406

[De79] Denny, W.M., et al., "Logic Enhanced Memories : An Overview and Some Examples of Their Applications to a Radar Tracking Problem", 1979, pp173-186

[Ep80] Epstein, R. and Hawthorn, P. B., "Design decisions for the intelligent database machine," *Proc. of National Computer Conference*, May 1980.

[Fo80] Foster, M.J., and Kung, H.T. "Design of Special Purpose VLSI Chips: Examples and Opinions." Proceedings of the seventh Annual Symposium of Computer Architecture (1980): 300-307

[Ha79] Hawthorn, P. B. and Stonebreaker, M., "Performance analysis of a relational database management system," *Proc. ACM-SIGMOD 1979 Int. Conf. Management of Data*, May 1979, pp. 1-12.

[Ha82] Hawthorn, P. B. and DeWitt, D. J., "Performance analysis of alternative database machine architectures," *IEEE Trans. on Software*

Engineering 8, 1 (January 1982), pp. 61-75.

[Le79] Leiserson C.E., " Systolic Priority Queues." Proceedings of Cal tech Conference of VLSIO (1979): 199-214

[Ma80] Maryanski, F. J., "Backend database systems," *Computing Surveys 12,* 1 (March 1980), pp. 3-25.

[Me80] C. Mead and L. Conway, Introduction to VLSI Systems, Addison - Wesley, Reading, Mass., 1980

[Mo79] Moore, G. "VLSI: Some Fundamental Challenges. " IEEE Spectrum, April, 1979

[Ou81] Ousterhout J.K., "Caesar: An Interactive Editor for VLSI Layout," VLSI Design, Fourth Quarter , 1981

[Ou83] Ousterhout J.K. , "Crystal: A Timing Analyzer for nMOS VLSI Circuits," Proc, Third Cal Tech. Cont. VLSI, Computer Science Press, Rockville, Md., 1983

[Ra78] Ramamoorthy, C.V., Ho, G.S. "A Design Methodology for User Oriented Computer Systems." NCC, 1978, pp953-966

[Ri80] Rideout, V. Leo, : Limits to Improvement of Silicon Integrated Circuits." Proceedings of Compcon (1980), 2-6

[Ro84] Nicholas Roussopoulos and Raymond T. Yeh, "An Adaptable Methodology for Database Design", *IEEE Computer,* May 1984.

[St76] Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES", *ACM Transactions on Database System,* Sep. 1976, pp.189-222.

[St81] Stonebreaker, M., "Operating system support for database management," *Comm. of ACM 24,* 7 (July 1981), pp. 412-418.

[Su84] Su, S. Y. W., *Database Machines: Concepts, Architectures and Techniques,* in progress.

[Su82] Suskind J.M. et al., " Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," in Proc. Conf. Advanced Research in VLSI, Paul Penfield, Jr., E., Artech House, Dedham, Mass., 1982

END

FILMED

DTIC